User's Guide for MS-BASIC

1
2 . 2
3 6
4 24
5 120

VICTOR®

VICTOR Publications
380 El Pueblo Road
Scotts Valley, CA 95066
(408) 438-6680

First VICTOR printing December, 1982.

PART NUMBER MS-8101-530-08 Rev. B Printed in U.S.A.

Contents

Using these Manuals

The information in the documents you received in this package is divided into reference information and user information.

The Microsoft BASIC Reference Manual contains descriptions of all the features of Microsoft BASIC. The Reference Manual contains no information that is either implementation specific (that is, applies to a particular microprocessor), or operating system specific.

The Microsoft BASIC User's Guide contains all of the implementation-specific and operating-system specific information. This information includes telling you which features of Microsoft BASIC are included and which are excluded from your implementation, telling you how your implementation changes the format and use of some features, and telling you how your operating system affects some features and operations of Microsoft BASIC.

Use the Reference Manual for details of the features. Use the User's Guide to see if a feature differs from its description in the Reference Manual. Also, use the User's Guide for hints about disk I/O and calling assembly language subroutines into your BASIC programs.

The Microsoft BASIC Reference Book is a quick reference guide to the features and their syntax. The Reference Book includes all the features described in the Reference Manual, but does not contain information that is implementation-specific or operating system-specific.

1. Four variable types: Integer (±32767), String (up to 255 characters), Single Precision Floating Point (7 digits), Double Precision Floating Point (16 digits)

2. Trace facilities (TRON/TROFF) for easier debugging

3. Error trapping using the ON ERROR GOTO statement

4. PEEK and POKE statements to read and write any memory location

5. Automatic line number generation and renumbering, including referenced line numbers

6. Arrays with up to 8 dimensions

7. Boolean operators OR, AND, NOT, XOR, EQV, IMP

8. Formatted output using the complete PRINT USING facility, including asterisk fill, floating dollar sign, scientific notation, trailing sign, comma insertion

9. Direct access to the 64K I/O ports with the INP and OUT functions

10. Extensive program editing facilities via EDIT command and EDIT mode subcommands.

11. Assembly language subroutine calls (up to 10 per program) are supported.

12. IF/THEN/ELSE and nested IF/THEN/ELSE constructs

13. Disk BASIC supports variable length random and sequential disk files with a complete set of file manipulation statements: OPEN, CLOSE, GET, PUT, KILL, NAME, MERGE

Syntax Notation

The following notation is used throughout this manual in descriptions of command and statement syntax:

- `[ ]` Square brackets indicate that the enclosed entry is optional.
- `< >` Angle brackets indicate user entered data. When the angle brackets enclose lower case text, the user must type in an entry defined by the text; for example, <filename>. When the angle brackets enclose upper case text, the user must press the key named by the text; for example, <RETURN>.
- `{ }` Braces indicate that the user has a choice between two or more entries. At least one of the entries enclosed in braces must be chosen unless the entries are also enclosed in square brackets.
- `...` Ellipses indicate that an entry may be repeated as many times as needed or desired.
- CAPS Capital letters indicate portions of statements or commands that must be entered, exactly as shown.

All other punctuation, such as commas, colons, slash marks, and equal signs, must be entered exactly as shown.

CHAPTER 1

FEATURES INCLUDED IN THIS IMPLEMENTATION

All statements, commands, and functions described in the Microsoft BASIC Reference Manual are implemented unless listed below. Some of the statements, commands, or functions may be affected by this particular implementation. See Chapter 2, Language Differences for This Implementation.

The following features are not included in the 8086 Version of Microsoft BASIC:

CLOAD CSAVE

The following statement is included in the 8086 version of Microsoft BASIC:

DEF SEG

DEF SEG and changes to some other statements and functions are described in Chapter 2, below.

CHAPTER 2

LANGUAGE DIFFERENCES FOR THIS IMPLEMENTATION

The following features differ in this implementation from the descriptions given in the Microsoft BASIC Reference Manual.

2.1 THE DEF SEG STATEMENT

The DEF SEG statement is not included in the Microsoft BASIC Reference Manual.

Format: DEF SEG [=<address>]

where: address is a valid numeric expression returning an unsigned integer in the range 0 to 65535.

Purpose: To assign the current segment address to be referenced by a subsequent CALL (see Section 2.2), a USR function call, or a PEEK or POKE statement.

Remarks: The address specified is saved for use as the segment required by PEEK, POKE, and CALL statements.

Entry of any value outside the <address> range 0-65535 will result in an "Illegal Function Call" error, and the previous value will be retained.

If the <address> option is omitted, the segment to be used is set to BASIC's data segment (DS). This is the initial default value.

If the <address> option is given, it should be based on a 16-byte boundary. For PEEK, POKE, or CALL statements, the value is shifted left 4 bits (this is done by the microprocessor, not by BASIC) to form the code segment address for the

subsequent call instruction. BASIC does not perform additional checking to assure that the resultant segment address is valid.

NOTE: DEF and SEG MUST be separated by a space. Otherwise, BASIC would interpret the statement DEFSEG=100 to mean, "assign the value 100 to the variable DEFSEG."

Example:
```
10 DEF SEG=&HB800 'Set segment to Screen buffer
20 DEF SEG 'Restore segment to BASIC's DS
```

2.2 THE CALL 'STATEMENT

Format: CALL <variable name> [(<argument list>)]

where: variable name contains the segment offset that is the starting point in memory of the subroutine being CALLed. Note that the variable name must be assigned to the segment offset before the CALL statement is issued (see example below).

argument list contains the variables or constants, separated by commas, that are to be passed to the routine.

Purpose: To call an assembly language subroutine.

Remarks: The CALL statement is the recommended way of calling 8086 machine language programs with Microsoft BASIC. It is suggested that the old style user-call USR(n) not be used. See Chapter 5 for comparison of the two methods and for a complete description of using the CALL statement for assembly language subroutines.

When a CALL statement is executed, control is transferred to the user's routine via the segment address given in the last DEF SEG statement and the segment offset specified by the <variable name> portion of the CALL statement. Values are returned to BASIC by including the variable name which will receive the result in the <argument list>.

The CALL statement conforms to the INTEL PL/M-86 calling conventions outlined in Chapter 9 of the INTEL PL/M-86 Compiler Operator's Manual. Microsoft BASIC follows the rules described for the MEDIUM case.

Example:

```
100 DEF SEG=&H8000
110 FOO=&H7FA
120 CALL FOO(A,B$,C)
      .
      .
      .
```

Line 100 sets the segment address to 8000 Hex. The variable FOO is set to &H7FA, so that the call to FOO will execute the subroutine at location 8000:7FA Hex (absolute address 807FA Hex).

2.3 THE INP FUNCTION

Format: INP(I)

where: I is a valid machine port number in the range 0 to 65535.

Purpose: To return the byte read from port I.

Remarks: INP is the complementary function to the OUT statement (Section 2.4 below).

Example: 100 A=INP(54321)

In assembly language, this is equivalent to:

```
MOV  DX,54321
IN   AL,DX
```

2.4 THE OUT STATEMENT

Format: OUT I,J

where: I and J are integer expressions in the range 0 to 65535. I is a machine port number, and J is the data to be transmitted.

Purpose: To send a byte to a machine output port.

Remarks: OUT is the complementary statement to the INP function (Section 2.3 above).

Example: 100 OUT 12345,225

In assembly language, this is equivalent to:

```
MOV  DX,12345
MOV  AL,255
OUT  DX,AL
```

2.5 USR FUNCTION CALLS

Although the CALL statement is recommended for calling assembly language subroutines, the USR function call may also be used. See Chapter 5 for comparison of CALL and USR and for a detailed discussion of calling assembly language subroutines.

Format: USR[<digit>][(argument)]

where: digit specifies which USR routine is being called. (See the Microsoft BASIC Reference Manual, Section 2.13, DEF USR, for the rules governing <digit>.) If <digit> is omitted, USR0 is assumed.

argument is any numeric or string expression.

Purpose: To call an assembly language subroutine.

Remarks: In this implementation, if a segment other than the default segment (BASIC's data segment, DS) is to be used, a DEF SEG statement MUST be executed prior to a USR call. This assures that the code segment points to the subroutine being called. The segment address given in the DEF SEG statement determines the starting segment of the subroutine. (See the DEF SEG statement in Section 2.1 above.)

For each USR function, a corresponding DEF USR statement must have been executed to define the USR call offset. This offset and the currently active DEF SEG segment address determine the starting address of the subroutine.

Example:

```
100 DEF SEG=&H8000
110 DEF USR0=0
120 X=5 'Note that X is single precision
130 Y = USR0(X)
140 PRINT Y
```

Note that the type (numeric or string) of the variable receiving the function call must be consistent with the argument passed; or it may be set to Integer by calling MAKINT in the user's routine before returning to BASIC.

CHAPTER 3

CONVERTING PROGRAMS TO MICROSOFT BASIC

If you have programs written in a BASIC other than Microsoft BASIC, some minor adjustments may be necessary before running them with Microsoft BASIC. Here are some specific things to look for when converting BASIC programs.

3.1 STRING DIMENSIONS

Delete all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the Microsoft BASIC statement DIM A$(J).

Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for Microsoft BASIC string concatenation.

In Microsoft BASIC, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Forms such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from position I to position J, must be changed as follows:

Other BASIC	Microsoft BASIC
X$=A$(I)	X$=MID$(A$,I,1)
X$=A$(I,J)	X$=MID$(A$,I,J-I+1)

If the substring reference is on the left side of an assignment and X$ is used to replace characters in A$, convert as follows:

Other BASIC	Microsoft BASIC
A$(I)=X$	MID$(A$,I,1)=X$
A$(I,J)=X$	MID$(A$,I,J-I+1)=X$

3.2 MULTIPLE ASSIGNMENTS

Some BASICs allow statements of the form:

```
10 LET B=C=0
```

to set B and C equal to zero. Microsoft BASIC would interpret the second equal sign as a logical operator and set B equal to -1 if C equaled 0. Instead, convert this statement to two assignment statements:

```
10 C=0:B=0
```

3.3 MULTIPLE STATEMENTS

Some BASICs use a backslash (\) to separate multiple statements on a line. With Microsoft BASIC, be sure all statements on a line are separated by a colon (:).

3.4 MAT FUNCTIONS

Programs using the MAT functions available in some BASICs must be rewritten using FOR...NEXT loops to execute properly.

CHAPTER 4

MICROSOFT BASIC DISK I/O

Disk I/O procedures for the beginning BASIC user are examined in this chapter. If you are new to BASIC or if you're getting disk related errors, read through these procedures and program examples to make sure you're using all the disk statements correctly.

Wherever a filename is required in a disk command or statement, use a name that conforms to your operating system's requirements for filenames. The MS-DOS operating system will append a default extension .BAS to the filename given in a SAVE, RUN, MERGE or LOAD command.

4.1 PROGRAM FILE COMMANDS

The following list reviews the commands and statements used in program file manipulation.

SAVE <filename>[,A] — Writes to disk the program that is currently residing in memory. Optional A writes the program as a series of ASCII characters. (Otherwise, BASIC uses a compressed binary format.)

(See Section 4.2 below for discussion of how to SAVE protected files.)

LOAD <filename>[,R] — Loads the program from disk into memory. Optional R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before LOADing. If R is included, however, open data files are kept open. Thus programs can be chained or loaded in sections and access the same data files. (LOAD <filename>,R and RUN <filename>,R are equivalent.)

RUN <filename>[,R] — RUN <filename> loads the program from disk into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open. (RUN <filename>,R and LOAD <filename>,R are equivalent.)

MERGE <filename> — Loads the program from disk into memory but does not delete the current contents of memory. The program line numbers on disk are merged with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE command, the "merged" program resides in memory, and BASIC returns to command level.

KILL <filename> — Deletes the file from the disk. <filename> may be a program file, or a sequential or random access data file.

NAME <old filename> AS<new filename> — To change the name of a disk file, execute the NAME statement, NAME <oldfile> AS <newfile>. NAME may be used with program files, random files, or sequential files.

4.2 PROTECTED FILE

If you wish to save a program in an encoded binary format, use the "Protect" option with the SAVE command. For example:

```
SAVE "MYPROG",P
```

A program saved this way cannot be listed or edited. You may also want to save an unprotected copy of the program for listing and editing purposes.

4.3 DISK DATA FILES - SEQUENTIAL AND RANDOM I/O

There are two types of disk data files that may be created and accessed by a BASIC program: sequential files and random access files.

4.3.1 Sequential Files

Sequential files are easier to create than random files but are limited in flexibility and speed when it comes to accessing the data. The data that is written to a sequential file is a series of ASCII characters stored, one item after another (sequentially), in the order it is sent. It is read back in the same way.

The statements and functions that are used with sequential files are:

OPEN	PRINT#	INPUT#	WRITE#
	PRINT# USING	LINE INPUT#	
CLOSE	EOF	LOC	

The following program steps are required to create a sequential file and access the data in the file:

1.	OPEN the file in "O" mode.	OPEN "O",#1,"DATA"
2.	Write data to the file using the PRINT# statement. (WRITE# may be used instead.)	PRINT#1,A$;B$;C$
3.	To access the data in the file, you must CLOSE the file and reOPEN it in "I" mode.	CLOSE #1 OPEN "I",#1,"DATA"
4.	Use the INPUT# statement to read data from the sequential file into the program.	INPUT#1,X$,Y$,Z$

Program 1 is a short program that creates a sequential file, "DATA", from information you input at the terminal:

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$="DONE" THEN END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 PRINT#1,N$;",";D$;",";H$
60 PRINT:GOTO 20
RUN
NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? etc.
```

PROGRAM 1 - CREATE A SEQUENTIAL DATA FILE

Now look at Program 2. It accesses the file "DATA" that was created in Program 1 and displays the name of everyone hired in 1978:

```
10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
EBENEEZER SCROOGE
SUPER MANN
Input past end in 20
Ok
```

PROGRAM 2 - ACCESSING A SEQUENTIAL FILE

Program 2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an "Input past end" error. To avoid getting this error, insert line 15 which uses the EOF function to test for end-of-file:

```
15 IF EOF(1) THEN END
```

and change line 40 to GOTO 15.

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement

```
PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to disk without explicit delimiters. The comma at the end of the format string serves to separate the items in the disk file.

The LOC function, when used with a sequential file, returns the number of sectors that have been written to or read from the file since it was OPENed.

4.3.1.1 Adding Data To A Sequential File -

If you have a sequential file residing on disk and later want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in "O" mode, you destroy its current contents. The following procedure can be used to add data to an existing file called "NAMES":

1. OPEN "NAMES" in "I" mode.
2. OPEN a second file called "COPY" in "O" mode.
3. Read in the data in "NAMES" and write it to "COPY".
4. CLOSE "NAMES" and KILL it.
5. Write the new information to "COPY".
6. Rename "COPY" as "NAMES" and CLOSE.
7. Now there is a file on disk called "NAMES" that includes all the previous data plus the new data you just added.

Program 3 illustrates this technique. It can be used to create or add onto a file called NAMES. This program also illustrates the use of LINE INPUT# to read strings with embedded commas from the disk file. Remember, LINE INPUT# will read in characters from the disk until it sees a carriage return (it does not stop at quotes or commas) or until it has read 255 characters.

```
10 ON ERROR GOTO 2000
20 OPEN "I",#1,"NAMES"
30 REM IF FILE EXISTS, WRITE IT TO "COPY"
40 OPEN "O",#2,"COPY"
50 IF EOF(1) THEN 90
60 LINE INPUT#1,A$
70 PRINT#2,A$
80 GOTO 50
90 CLOSE #1
100 KILL "NAMES"
110 REM ADD NEW ENTRIES TO FILE
120 INPUT "NAME";N$
130 IF N$="" THEN 200 'CARRIAGE RETURN EXITS INPUT LOOP
140 LINE INPUT "ADDRESS? ";A$
150 LINE INPUT "BIRTHDAY? ";B$
160 PRINT#2,N$
170 PRINT#2,A$
180 PRINT#2,B$
190 PRINT:GOTO 120
200 CLOSE
205 REM CHANGE FILENAME BACK TO "NAMES"
210 NAME "COPY" AS "NAMES"
2000 IF ERR=53 AND ERL=20 THEN OPEN "O",#2,"COPY":RESUME 120
2010 ON ERROR GOTO 0
```

PROGRAM 3 - ADDING DATA TO A SEQUENTIAL FILE

The error trapping routine in line 2000 traps a "File does not exist" error in line 20. If this happens, the statements that copy the file are skipped, and "COPY" is created as if it were a new file.

4.3.2 Random Files

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using random files. One advantage is that random files require less room on the disk, because BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage to random files is that data can be accessed randomly, i.e., anywhere on the disk -- it is not necessary to read through all the information, as with sequential files. This is possible because the information is stored and accessed in distinct units called records and each record is numbered.

The statements and functions that are used with random files are:

OPEN	FIELD	LSET/RSET	GET
PUT	CLOSE	LOC	
MKI$	CVI		
MKS$	CVS		
MKD$	CVD		

4.3.2.1 Creating A Random File -

The following program steps are required to create a random file.

1. OPEN the file for random access ("R" mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.

```
OPEN "R",#1,"FILE",32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file.

```
FIELD #1, 20 AS N$,
 4 AS A$, 8 AS P$
```

3. Use LSET to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ for a single precision value, and MKD$ for a double precision value.

```
LSET N$=X$
LSET A$=MKS$(AMT)
LSET P$=TEL$
```

4. Write the data from the buffer to the disk using the PUT statement.

```
PUT #1,CODE%
```

Look at Program 4. It takes information that is input at the terminal and writes it to a random file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

NOTE

Do not use a FIELDed string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 INPUT "NAME";X$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$:PRINT
70 LSET N$=X$
80 LSET A$=MKS$(AMT)
90 LSET P$=TEL$
100 PUT #1,CODE%
110 GOTO 30
```

PROGRAM 4 - CREATE A RANDOM FILE

4.3.2.2 Accessing A Random File -

The following program steps are required to access a random file:

1. OPEN the file in "R" mode.

```
OPEN "R",#1,"FILE",32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file.

```
FIELD #1 20 AS N$,
       4 AS A$, 8 AS P$
```

NOTE

In a program that performs both input and output on the same random file, you can often use just one OPEN statement and one FIELD statement.

3. Use the GET statement to move the desired record into the random buffer.

```
GET #1,CODE%
```

4. The data in the buffer may now be accessed by the program. Numeric values must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single precision values, and CVD for double precision values.

```
PRINT N$
PRINT CVS(A$)
```

Program 5 accesses the random file "FILE" that was created in Program 4. By inputting the three-digit code at the terminal, the information associated with that code is read from the file and displayed:

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

PROGRAM 5 - ACCESS A RANDOM FILE

The LOC function, with random files, returns the "current record number." The current record number is one plus the last record number that was used in a GET or PUT statement. For example, the statement

```
IF LOC(1)>50 THEN END
```

ends program execution if the current record number in file#1 is higher than 50.

Program 6 is an inventory program that illustrates random file access. In this program, the record number is used as the part number, and it is assumed the inventory will contain no more than 100 different part numbers. Lines 900-960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

```
120 OPEN"R",#1,"INVEN.DAT",39
125 FIELD#1,1 AS F$,30 AS D$, 2 AS Q$,2 AS R$,4 AS P$
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT 1,"INITIALIZE FILE"
140 PRINT 2,"CREATE A NEW ENTRY"
150 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER LEVEL"
220 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
225 IF (FUNCTION<1)OR(FUNCTION>6) THEN PRINT
        "BAD FUNCTION NUMBER":GO TO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680
240 GOTO 220
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT"OVERWRITE";A$:
        IF A$<>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
360 LSET P$=MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$##.##";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD ";A%
```

```
520 Q%=CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q%=CVI(Q$)
620 IF (Q%-S%)<0 THEN PRINT "ONLY";Q%;" IN STOCK":GOTO 600
630 Q%=Q%-S%
640 IF Q%=<CVI(R$) THEN PRINT "QUANTITY NOW";Q%;
        " REORDER LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710 GET#1,I
720 IF CVI(Q$)<CVI(R$) THEN PRINT D$;" QUANTITY";
          CVI(Q$) TAB(50) "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF(PART%<1)OR(PART%>100) THEN PRINT "BAD PART NUMBER":
        GOTO 840 ELSE GET#1,PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B$:IF B$<>"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

PROGRAM 6 - INVENTORY

CHAPTER 5

BASIC ASSEMBLY LANGUAGE SUBROUTINES

All versions of Microsoft BASIC have provisions for interfacing with assembly language subroutines via the USR function and the CALL statement.

The USR function allows assembly language subroutines to be called in the same way BASIC Intrinsic functions are called. However, the CALL statement is the recommended way of interfacing 8086 machine language programs with BASIC. It is compatible with more languages than is the USR function call, it produces more readable source code, and it can pass multiple arguments.

5.1 MEMORY ALLOCATION

IMPORTANT

Memory space must be set aside for an assembly language subroutine before it can be loaded. During initialization, enter the highest memory location minus the amount of memory needed for the assembly language subroutine(s) with the /M: switch.

In addition to the BASIC interpreter code area, Microsoft BASIC uses up to 64K of memory beginning at its data (DS) segment.

If, when an assembly language subroutine is called, more stack space is needed, BASIC's stack can be saved and a new stack set up for use by the assembly language subroutine. BASIC's stack must be restored, however, before returning from the subroutine.

The assembly language subroutine may be loaded into memory by means of the operating system or the BASIC POKE statement. If the user has the Microsoft Utility Software Package, the routines may be assembled with the MACRO-86

assembler and linked using the MS-LINK Linker, but not loaded. To load the program file, the user should observe these guidelines:

The subroutines must not contain any long references.

Skip over the first 512 bytes of the MS-LINK output file, then read in the rest of the file.

5.2 THE CALL STATEMENT

As mentioned earlier, the CALL statement is the recommended way of interfacing 8086 machine language programs with BASIC. It is further suggested that the old style user-call USR(n) not be used.

Format:

CALL <variable name> [(<argument list>)]

where: variable name contains the segment offset that is the starting point in memory of the subroutine being CALLed.

argument list contains the variables or constants, separated by commas, that are to be passed to the routine.

The CALL statement conforms to the INTEL PL/M-86 calling conventions outlined in Chapter 9 of the INTEL PL/M-86 Compiler Operator's Manual. BASIC follows the rules described for the MEDIUM case (summarized in the following discussion).

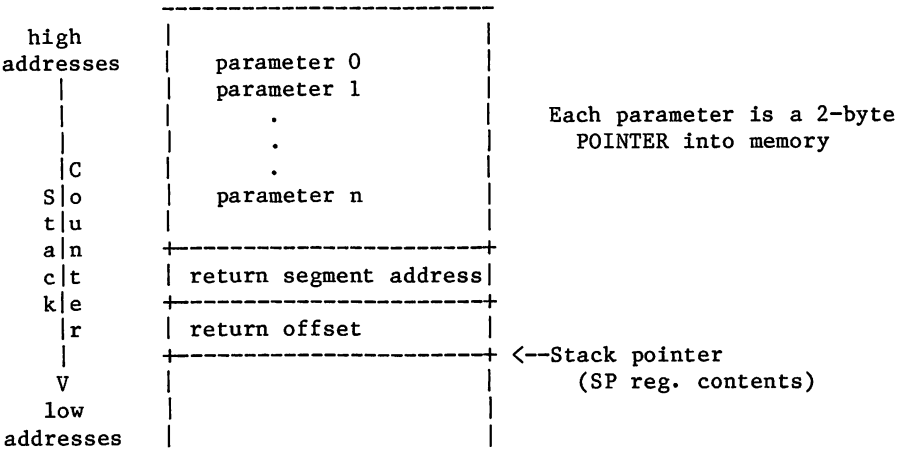
Invoking the CALL statement causes the following to occur:

1. For each parameter in the argument list, the 2 byte offset of the parameter's location within the data segment (DS) is pushed onto the stack.

2. BASIC's return address code segment (CS), and offset (IP) are pushed onto the Stack.

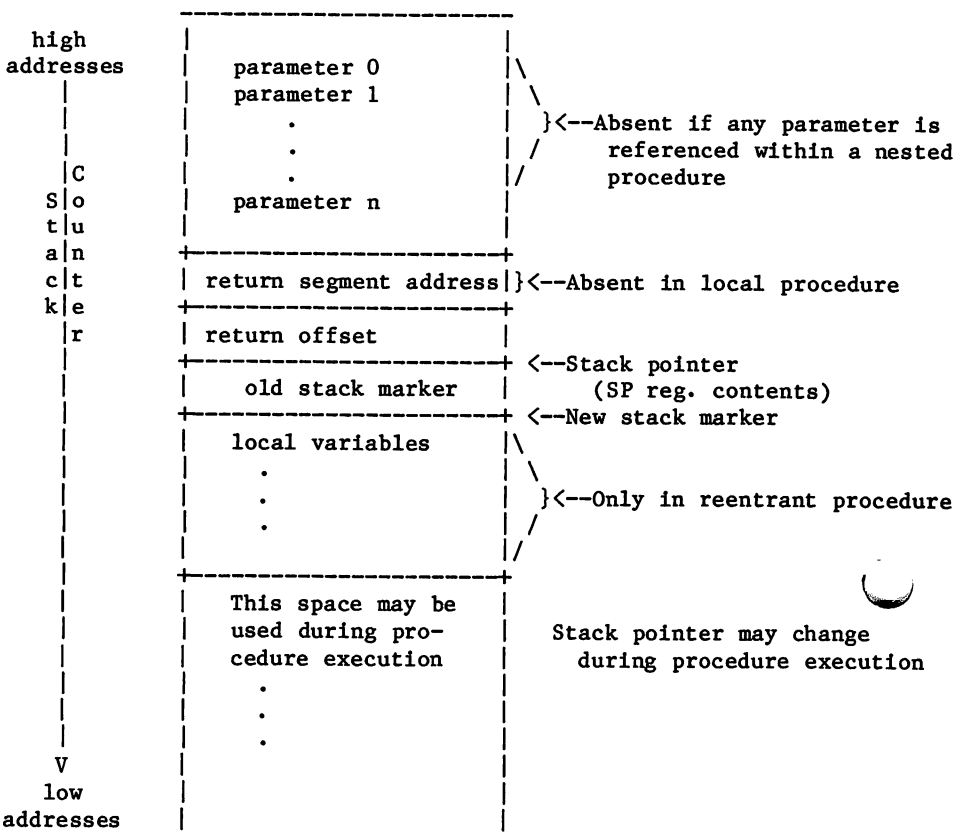
3. Control is transferred to the user's routine via an 8086 long call to the segment address given in the last DEF SEG statement and the offset given in <variable name>.

These actions are illustrated by the two following diagrams, which illustrate first, the state of the stack at the time of the CALL statement, and second, the condition of the stack during execution of the CALLed subroutine.



Stack Layout when CALL statement is activated

The user's routine now has control. Parameters may be referenced by moving the Stack pointer (SP) to the Base Pointer (BP) and adding a positive offset to (BP).



Stack Layout during Execution of
of a CALL statement

You must observe the following rules when coding a subroutine:

1. The CALLed routine may destroy the AX, BX, CX, DX, SI, DI, and BP registers.

2. The CALLed program MUST know the number and length of the parameters passed. References to parameters are positive offsets added to (BP) (assuming the called routine moved the current stack pointer into BP; i.e., MOV BP,SP). That is, the location of p1 is at 8(BP), p2 is at 6(BP), p3 is at 4(BP), ...etc.

3. The CALLed routine must do a RET <n> (where <n> is two times the number of parameters in the argument list) to adjust the stack to the start of the calling sequence.

4. Values are returned to BASIC by including in the argument list the variable name which will receive the result.

5. If the argument is a string, the parameter's offset points to 3 bytes called the "String Descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

IMPORTANT

If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add +"" to the string literal in the program. Example:

```
20 A$ = "BASIC"+""
```

This will force the string literal to be copied into string space. Now the string may be modified without affecting the program.

6. Strings may be altered by user routines, but the length MUST NOT be changed. BASIC cannot correctly manipulate strings if their lengths are modified by external routines.

Example:

```
100 DEF SEG=&H8000
110 FOO=&H7FA
120 CALL FOO(A,B$,C)
      .
      .
      .
```

Line 100 sets the segment to 8000 Hex. The value of variable FOO is added into the address as the low word after the DEF SEG value is left shifted 8 bits. (This is a function of the microprocessor, not of BASIC.) Here, FOO is set to &H7FA, so that the call to FOO will execute the subroutine at location 8000:7FA Hex (absolute address 807FA Hex).

The following sequence of 8086 assembly language demonstrates access of the parameters passed and storing a return result in the variable 'C'.

```
MOV     BP,SP       ;Get current Stack posn in BP.
MOV     BX,6[BP]    ;Get address of B$ dope.
MOV     CL,[BX]     ;Get length of B$ in CL.
MOV     DX,1[BX]    ;Get addr of B$ text in DX.
        .
        .
        .
MOV     SI,8[BP]    ;Get address of 'A' in SI.
MOV     DI,4[BP]    ;Get pointer to 'C' in DI.
MOVS    WORD        ;Store variable 'A' in 'C'.
RET     6           ;Restore Stack, return.
```

IMPORTANT

The called program must know the variable type for numeric parameters passed. In the above example, the instruction

MOVS WORD

will copy only 2 bytes. This is fine if variables A and C are integer. We would have to copy 4 bytes if they were Single Precision and copy 8 bytes if they were Double Precision.

5.3 USR FUNCTION CALLS

Although the CALL statement is the recommended way of calling assembly language subroutines, the USR function call is still available for compatibility with previously-written programs.

The format of the USR function call is:

USR[<digit>][(argument)]

where: digit is from 0 to 9. <digit> specifies which USR routine is being called (see DEF USR statement, Section 2.13 of the Reference Manual). If <digit> is omitted, USR0 is assumed.

argument is any numeric or string expression. Arguments are discussed in detail below.

In this implementation of BASIC, a DEF SEG statement MUST be executed prior to a USR call to assure that the code segment points to the subroutine being called. The segment address given in the DEF SEG statement determines the starting segment of the subroutine. (See the DEF SEG statement in Chapter 2.)

For each USR function, a corresponding DEF USR statement must have been executed to define the USR call offset. This offset and the currently active DEF SEG address determine the starting address of the subroutine.

When the USR function call is made, register [AL] contains a value which specifies the type of argument that was given. The value in [AL] may be one of the following:

2 Two-byte integer (two's complement)

3 String

4 Single precision floating point number

8 Double precision floating point number

If the argument is a number, the [BX] register pair points to the Floating Point Accumulator (FAC) where the argument is stored:

FAC is the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa.

FAC-1 contains the highest 7 bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0=positive, 1=negative).

If the argument is an integer:

FAC-2 contains the upper 8 bits of the argument.

FAC-3 contains the lower 8 bits of the argument.

If the argument is a single precision floating point number:

FAC-2 contains the middle 8 bits of mantissa.

FAC-3 contains the lowest 8 bits of mantissa.

If the argument is a double precision floating point number:

FAC-7 - FAC-4 contain four more bytes of mantissa (FAC-7 contains the lowest 8 bits).

If the argument is a string:

the [DX] register pair points to 3 bytes called the "string descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in BASIC's Data Segment.

IMPORTANT

If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. See the CALL statement above.

Usually, the value returned by a USR function is the same type (integer, string, single precision, or double precision) as the argument that was passed to it.

Example:

```
110 DEF USR0=&H8000 'Assumes user gave /M:32767
120 X=5 'Note that X is single precision
130 Y = USR0(X)
140 PRINT Y
```

Note that the type (numeric or string) of the variable receiving the function call must be consistent with the argument passed; or it may be set to Integer by calling MAKINT in the user's routine before returning to BASIC.

We have loaded the following assembly language routine to simply multiply the argument passed by 2 and return an integer result.

Always be sure that your programs are defined by a PROC FAR statement.

```
DOUBLE          SEGMENT

                ASSUME  CS:DOUBLE

FRCINTOFFSET    EQU     103H
MAKINTOFFSET    EQU     107H

FRCINT          LABEL   DWORD
                DW      FRCINTOFFSET
FRCSEG          DW      ?

MAKINT          LABEL   DWORD
                DW      MAKINTOFFSET
MAKSEG          DW      ?

USRPRG          PROC    FAR
                POP     SI
                POP     AX          ;Recover BASIC's CS
                PUSH    AX
                PUSH    SI
                MOV     FRCSEG,AX   ;Set segment for long indirect CALL
                MOV     MAKSEG,AX
                CALL    FRCINT      ;Force arg in FAC to int in [BX]
                ADD     BX,BX       ; [BX] = [BX] * 2
                CALL    MAKINT      ;Put Result back in FAC
                RET                 ;Long return to BASIC
USRPRG          ENDP

DOUBLE          ENDS
```

When FRCINT or MAKINT is called and when the subroutine terminates with a return, ES, DS, and SS must have the same

value they had when the subroutine was entered. These registers point to BASIC's Data Segment.

CHAPTER 6

MICROSOFT BASIC WITH THE MS-DOS OPERATING SYSTEM

The MS-DOS version of Microsoft BASIC is supplied on a standard size 3740 single density diskette. The name of the file is BASIC86.COM. (A 48K or larger MS-DOS system is recommended.)

To run BASIC, bring up MS-DOS and type the following:

A>BASIC86 <carriage return> MSBASIC.COM

The system will reply:

```
xxxx Bytes Free
Microsoft BASIC Version 5.x
[MS-DOS Version]
Copyright 1977-1981 (C) by Microsoft, Inc.
Created: dd-mmm-yy
Ok
```

6.1 INITIALIZATION

The initialization dialog has been replaced by a set of options which are placed after the BASIC86 command to MS-DOS. The format of the command line is (the command line may appear differently on your screen than on this page):

```
A>BASIC86 [<filename>][/F:<number of files>]
       [/M:<highest memory location>][/S:<maximum record size>]
```

If <filename> is present, BASIC proceeds as if a RUN <filename> command were typed after initialization is complete. A default extension of .BAS is used if none is supplied and the filename is less than 9 characters long. This allows BASIC programs to be executed in batch mode using the BATCH facility of MS-DOS. Such programs should include a SYSTEM statement (see below) to return to MS-DOS when they have finished, allowing the next program in the batch stream to execute.

If /F:<number of files> is present, it sets the number of disk data files that may be open at any one time during the execution of a BASIC program. Each file data block allocated in this fashion requires 166 bytes of memory. If the /F option is omitted, the number of files defaults to 3.

The /M:<highest memory location> option sets the highest memory location that will be used by BASIC.

/S:<maximum record size> may be added at the end of the command line to set the maximum record size for use with random files. The default record size is 128 bytes.

NOTE

<number of files>, <highest memory location>, and <maximum record size> are numbers that may be either decimal, octal (preceded by &O) or hexadecimal (preceded by &H).

Examples:

```
A>BASIC86 PAYROLL.BAS       Use all memory and 3 files,
                            load and execute PAYROLL.BAS.

A>BASIC86 INVENT/F:6        Use all memory and 6 files,
                            load and execute INVENT.BAS.

A>BASIC86 /M:32768          Use first 32K of memory and
                            3 files.

A>BASIC86 DATACK/F:2/M:&H9000
                            Use first 36K of memory, 2
                            files, and execute DATACK.BAS.
```

6.2 DISK FILES

Disk filenames follow the normal MS-DOS naming conventions. All filenames may include A: or B: as the first two characters to specify a disk drive; otherwise the currently selected drive is assumed. A default extension of .BAS is used on LOAD, SAVE, MERGE and RUN <filename> commands if no "." appears in the filename and the filename is less than 9 characters long.

Large random files are supported. The maximum logical record number is 32767. If a record size of 256 is specified, then files up to 8 megabytes can be accessed.

6.3 FILES COMMAND

Format: FILES[<filename>]

Purpose: To print the names of files residing on the current disk.

Remarks: If <filename> is omitted, all the files on the currently selected drive will be listed. <filename> is a string formula which may contain question marks (?) to match any character in the filename or extension. An asterisk (*) as the first character of the filename or extension will match any file or any extension.

Examples:
```
FILES
FILES "*.BAS"
FILES "B:*.*"
FILES "TEST?.BAS"
```

6.4 RESET COMMAND

Format: RESET

Purpose: To close all disk files and write the directory information to a diskette before it is removed from a disk drive.

Remarks: Always execute a RESET command before removing a diskette from a disk drive. Otherwise, when the diskette is used again, it will not have the current directory information written on the directory track.

RESET closes all open files on all drives and writes the directory track to every diskette with open files.

6.5 LOF FUNCTION

Format: LOF(<file number>)

Purpose: Returns the length of the file in bytes.

Example:
```
110 IF NUM%>LOF(1) THEN PRINT "INVALID ENTRY"
```

6.6 EOF

Format: EOF (<file number>)

Purpose: Returns -1 (true) if the end of a sequential or random file has been reached. If a GET is done past the end of file, EOF will return -1. This may be used to find the size of a file using a binary search or other algorithm.

Example:
```
10 OPEN "I",1,"DATA"
20 C=0
30 IF EOF(1) THEN 100
40 INPUT #1,M(C)
50 C=C+1:GOTO 30
```

6.7 MISCELLANEOUS

1. CSAVE and CLOAD are not implemented.

2. To return to MS-DOS, use the SYSTEM command or statement. SYSTEM closes all files and then returns to MS-DOS command. Control-C always returns to BASIC, not to MS-DOS.

3. FRCINT is at 103 hex and MAKINT is at 107 hex.

INDEX

Microsoft BASIC

Reference Manual

Microsoft BASIC
Reference Manual

Contents